1. Tempo disponibile 120 minuti (90 minuti per gli studenti di "Introduzione agli Algoritmi" - 6 CFU, che devono fare solo i primi 3 esercizi).
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY:

Algorithm 1: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← 0
    for i ← 1 to n do
        for j ← n downto 1 do
            v ← v + i − j
        end
    end
    return v + Mystery(n/2) − Mystery2(n)
end

function Mystery2(Int m) → Int
Int x ← Mystery(m/2)
return 2 × x − Mystery(m/2)
```

Soluzione. Indichiamo con $T(n)$ e $T'(m)$ i costi computazionali delle due invocazioni di funzioni MYSTERY(n) e MYSTERY2(m), rispettivamente. Visto che MYSTERY(n), se $n > 1$, esegue operazioni di costo costante all'interno di due for annidati di esattamente n cicli (oltre ad una chiamata ricorsiva con parametro $n/2$ ed una chiamata MYSTERY2(n)), e che MYSTERY2(m) effettua due chiamate MYSTERY($m/2$), abbiamo che:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + T'(n) + c_2 \times n^2 & \text{altrimenti} \end{cases} \qquad T'(m) = 2 \times T(m/2)$$

Come primo passaggio sostituiamo, nella prima relazione, $T'(n)$ con la relativa definizione, ottenendo:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + 2 \times T(n/2) + c_2 \times n^2 & \text{altrimenti} \end{cases}$$

Ora abbiamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3 \times T(n/2) + c_2 \times n^2 & \text{altrimenti} \end{cases}$$

che risolviamo tramite Master Theorem. Abbiamo $a = 3$, $b = 2$ e $\beta = 2$; quindi $\alpha = \frac{\log 3}{\log 2} = \log 3$, e per il terzo caso 3 del Master Theorem (in quanto $\log 3 < 2$) possiamo concludere $T(n) = \Theta(n^2)$.

2. Si scriva una procedura che, date due liste concatenate monodirezionali, L_1 e L_2, contenenti interi **ordinati** dal più piccolo al più grande, rimuova da L_1 tutti gli interi che appaiono in L_2 (senza modificare L_2). Esempio: se L_1 = [1,1, 2, 3,4,4] e L_2 = [1,3,3,5], al termine della procedura L_1 = [2,4,4] e L_2 = [1,3,3,5]. Discutere il costo computazionale della procedura proposta nel caso ottimo e nel caso pessimo.

Soluzione. Una possibile soluzione, non particolarmente efficiente, effettua una scansione delle lista L_1 per ogni intero presente in L_2 (vedi Algoritmo 2). La funzione ListComplement1 esegue una chiamata alla funzione Remove per ogni elemento in L_2. Nel caso pessimo, la funzione di supporto Remove ha un costo pari al numero di interi nella lista L, O($|L|$), mentre nel caso ottimo (quando x è minore di ogni intero in L) ha un costo costante, O(1). Quindi, nel caso pessimo ListComplement1 costa O($|L_1||L_2|$) e nel caso ottimo O($|L_2|$).

Algorithm 2: ListComplement1(List L_1, List L_2) → List

```
while L2 ≠ NULL do
    L1 ← Remove(L1,L2.val)
    L2 ← L2.next
end
return L1

function Remove(List L, Int x) → List
if L = NULL or x < L.val then
    return L
else if L.val = x then
    // Necessario deallocare L qui se il linguaggio non dispone di un garbage collector
    return Remove(L.next,x)
else
    L.next ← Remove(L.next,x)
    return L
```

La funzione ListComplement1 non sfrutta interamente l'ordinamento imposto a L_1 ed L_2. Possiamo implementare una funzione maggiormente efficiente, che evita di visitare L_1 dall'inizio per ogni valore in L_2: se il nodo correntemente visitato in L_1 ha un valore maggiore di quello del nodo correntemente visitato in L_2, avanziamo su L_2; diversamente, se il nodo corrente in L_1 ha un valore minore, avanziamo su L_1; se i due valori sono uguali, rimuoviamo il nodo corrente in L_1 e proseguiamo a valutare il nodo successivo in L_1 senza avanzare su L_2. Per implementare questa seconda soluzione preferiamo una versione interamente ricorsiva ListComplement2 (vedi Algoritmo 3). Nel caso pessimo ListComplement2 visita tutti gli elementi dei entrambe le liste (ad ogni step avanziamo o su L_1 oppure su L_2, mai su entrambe le liste contemporaneamente): costo O($|L_1| + |L_2|$). Nel caso ottimo tutti gli interi in L_1 sono minori degli interi in L_2 (o viceversa). In questo caso, la funzione scorre solo la lista con gli elementi minori: costo O($\min(|L_1|, |L_2|)$).

Cont.

Algorithm 3: ListComplement2(List L_1, List L_2) $\rightarrow$ List

```
if L1 = NULL or L2 = NULL then
    return L1
else if L1.val = L2.val then
    // Necessario deallocare L1 qui se il linguaggio non dispone di un garbage collector
    return ListComplement2(L1.next,L2)
else if L1.val > L2.val then
    return ListComplement2(L1,L2.next)
else
    L1.next ← ListComplement2(L1.next,L2)
    return L1
```

3. Bisogna preparare il trasporto di un oggetto fragile, che viene collocato in un cartone che deve poi essere riempito con pezzi di polistirolo per limitarne i possibili danni durante il trasporto. Il volume complessivo del polistirolo da inserire è un intero K. Esistono n diversi formati di pezzi di polistirolo, ognuno con un proprio volume $v[i]$, con $v[1..n]$ array di interi. Per ogni formato, sono disponibili una quantità arbitraria di pezzi da poter utilizzare. Per rendere più sicura la spedizione, si desidera massimizzare il numero di pezzi di polistirolo da inserire nel cartone. Progettare un algoritmo che dati K e $v[1..n]$ restituisce un array $x[1..n]$ che indica che, per ottenere il volume K massimizzando il numero di pezzi di polistirolo, si possono usare $x[i]$ pezzi del formato i-esimo. Nel caso in cui non sia possibile raggiungere il volume complessivo K con i formati di polistirolo disponibili, l'array x conterrà valori tutti uguali a 0 (ovvero, $x[i] = 0$ per tutti gli $i \in \{1..n\}$).

   Soluzione. È possibile procedere utilizzando programmazione dinamica, considerando i seguenti sottoproblemi: $P(i, j)$, con $i \in \{1..n\}$ e $j \in \{0..K\}$, che indica il numero massimo di pezzi di polistirolo delle prime i tipologie, che possono essere usati per ottenere esattamente un volume j. Nel caso in cui cui non sia possibile ottenere esattamente tale volume, poniamo $P(i, j) = -1$. Tali problemi possono essere risolti induttivamente rispetto a i nel seguente modo:

$$P(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j = 0 \\ -1 & \text{se } i = 1,\ j > 0 \text{ e } j\%v[i] \neq 0 \\ j/v[i] & \text{se } i = 1,\ j > 0 \text{ e } j\%v[i] = 0 \\ P(i-1,j) & \text{se } i > 1 \text{ e } j < v[i] \\ max\{P(i-1,j), 1 + P(i,j-v[i])\} & \text{altrimenti} \end{cases}$$

   Procediamo quindi a progettare un algoritmo (si veda Algoritmo 4) che risolve i problemi $P(i, j)$ memorizzando le relative soluzioni in una tabella $T[1..n, 0..K]$. Per poter poi ricostruire la combinazione di pezzi di polistirolo che genera la soluzione ottima, si utilizza una ulteriore tabella booleana $B[1..n, 0..K]$ tale che $B[i, j] = true$ se e solo se un pezzo di polistirolo di tipo i fa parte di una possibile soluzione ottima del problema $P(i, j)$. L'algoritmo proposto ha costo computazionale $T(K, n) = \Theta(n \times K)$.

Cont.

Algorithm 4: POLISTIROLO(INT K,INT $v[1..n]$) $\rightarrow$ INT[1..N]

```
// Inizializzazione prima riga delle tabelle T e B
T[1,0] ← 0; B[1,0] ← false
for j ← 1 to K do
    if j%p[1] ≠ 0 then
        T[1,j] ← −1; B[1,j] ← false
    else
        T[1,j] ← j/p[i]; B[1,j] ← true
    end
end
// Riempimento restanti righe delle tabelle T e B
for i ← 2 to n do
    for j ← 0 to K do
        if j < v[i] then
            T[i,j] ← T[i − 1,j]; B[i,j] ← false
        else
            T[i,j] ← 1 + T[i,j − v[i]]; B[i,j] ← true
    end
end
// Costruzione della soluzione ottimale x
for i ← 1 to n do
    x[i] ← 0
end
if T[n,K] ≠ −1 then
    // Almeno una soluzione ottima esiste
    i ← n; volumeRimanente ← K
    while volumeRimanente > 0 do
        if B[i,volumeRimanente] then
            x[i] ← x[i] + 1
            volumeRimanente ← volumeRimanente − v[i]
        else
            i ← i − 1
        end
    end
end
return x
```

4. Progettare un algoritmo che dato un grafo orientato pesato $G = (V, E, w)$ verifica se G contiene un ciclo di costo complessivo negativo.

   Soluzione. Tra gli algoritmi visti a lezione, è possibile utilizzare Floyd-Warshall e al termine della sua esecuzione controllare se è stata calcolata una distanza negativa fra un vertice e se stesso. Tale soluzione (si veda Algoritmo 5) ha un costo computazionale coincidente con il costo dell'algoritmo di Floyd-Warshall, ovvero $\Theta(|V|^3)$.

   Si noti che non è possibile utilizzare direttamente l'algoritmo di Bellman-Ford in quanto tale algoritmo considera un dato vertice di partenza ed è in grado di verificare la presenza di cicli di costo negativo raggiungibili da tale vertice. Quindi, se si esegue Bellman-Ford a partire da un vertice $v \in V$, ed esistono cicli di costo negativo che coinvolgono solo vertici non raggiungibili da v, tale algoritmo non sarà in grado di verificare la presenza di tali cicli.

Cont.

Algorithm 5: VerificaCicliNegativi(Grafo $G = (V, E, w) \rightarrow$ Bool

```
// esecuzione dell'algoritmo di Floyd-Warshall
n ← G.numNodi()
Real D[1..n, 1..n]
for x ← 1 to n do
    for y ← 1 to n do
        if x = y then
            D[x, y] = 0
        else if (x, y) ∈ E then
            D[x, y] = w(x, y)
        else
            D[x, y] = ∞
        end
    end
end
for k ← 1 to n do
    for x ← 1 to n do
        for y ← 1 to n do
            if D[x, k] + D[k, y] < D[x, y] then
                D[x, y] = D[x, k] + D[k, y]
            end
        end
    end
end
// verifica presenza cicli negativi
for v ∈ V do
    if D[v, v] < 0 then
        return true
    end
end
return false
```